Λειτουργικά Συστήματα Υπολογιστών
7ο Εξάμηνο, ΣΗΜΜΥ ΕΜΠ
26/1/2017

Αναφορά 4ης Άσκησης

Ομάδα Ε15
Γαβαλάς Νικόλαος 03113121
Καραΐσκου Κωνσταντίνα 03113127

Άσκηση 1.1

Source Code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */

/*  Global Variable για την τρέχουσα διεργασία */
pid_t current;

/*
*  Queue Initialization and Functions
*/
```

```c
typedef struct node_t
{
	pid_t pid;
	struct node_t *next;
} node;

node *front = NULL;
node *rear = NULL;

/* Για να ξέρουμε πότε η ουρά είναι άδεια */
int isEmpty(void)
{
	if(front == NULL) return 1;
	else return 0;
}

/* Τοποθετεί στην ουρά έναν νέο κόμβο με pid το όρισμα */
void enqueue(pid_t x)
{
	node *temp = (node *)malloc(sizeof(node));
	temp->pid = x;
	temp->next = NULL;
	if(front == NULL && rear == NULL)
	{
		front = rear = temp;
		return;
	}
	rear->next = temp;
	rear = temp;
}

/* Ελευθερώνει τον πρώτο κόμβο και επιστρέφει το pid αυτού */
pid_t dequeue(void)
{
	pid_t ret = 0;
	node *temp = front;
	if(front == NULL)
	{
		printf("Queue is empty!\n");
		return 0;  // Error value (Επειδή δουλεύουμε με pids,που είναι
	// θετικά, αν γυρίσει 0 έχουμε queue underflow)
```

```
        }
        if(front == rear)
        {
                ret = temp->pid;
                front = rear = NULL;
        }
        else
        {
                front = front->next;
                ret = temp->pid;
        }
        free(temp);
        return ret;
}

/* Συνάρτηση που τυπώνει τα περιεχόμενα της ουράς */
void print_queue()
{
        node *temp;
        printf("Queue values:\n");
        for(temp = front; temp != NULL; temp = temp->next)
        {
                printf("%d ", temp->pid);
        }
        printf("\n");
}

/*
* SIGALRM handler
*/
static void sigalrm_handler(int signum)
{
        // assert(0 && "Please fill me!");
        if(signum == SIGALRM)
        {
      /* Στο sigalrm απλά στέλνουμε sigstop στην τρέχουσα διεργασία */
                printf("Caught SIGALRM\n");
                kill(current, SIGSTOP);
                return;
        }
        else
```

```
	{
		printf("Unknown signal caught. Continuing...\n");
		return;
	}
}

/*
* SIGCHLD handler
*/
static void sigchld_handler(int signum)
{
	pid_t p;
	int status;
	for (;;)
	{
		p = waitpid(-1, &status, WUNTRACED | WNOHANG);
		if (p < 0)
		{
			perror("waitpid");
			exit(1);
		}
		if (p == 0)
			break;
		explain_wait_status(p, status);
		if (WIFEXITED(status) || WIFSIGNALED(status))
		{
		/* A child has died */
			printf("Child with pid %d died.\n", current);
			if(isEmpty())
			{
				printf("All done. Exiting...\n");
				exit(0);
			}
			else
			{
				current = dequeue();
				if(current == 0)
				{
					printf("Tried to dequeue from empty queue.
Exiting...");
					exit(0);
```

```
                                }
                            /* Όταν πεθάνει κάποιο παιδί, στέλνουμε SIGCONT στο επόμενο αφού πρώτα το πάρουμε από τη λίστα */
                                kill(current, SIGCONT);
                        }
                }
                if (WIFSTOPPED(status))
                {
            /* A child has stopped due to SIGSTOP/SIGTSTP, etc */
                        /* Όταν σταματήσει κάποιο παιδί, το βάζουμε στο τέλος της ουράς και στέλνουμε SIGCONT στο επόμενο αφού πριν το πάρουμε από τη λίστα */
                        enqueue(current);
                        current = dequeue();
                        kill(current, SIGCONT);
                        alarm(SCHED_TQ_SEC);
                }
        }
}

/* Install two signal handlers.
* One for SIGCHLD, one for SIGALRM.
* Make sure both signals are masked when one of them is running.
*/
static void
install_signal_handlers(void)
{
        sigset_t sigset;
        struct sigaction sa;

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

        sa.sa_handler = sigalrm_handler;
```

```
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
                exit(1);
        }

        /*
         * Ignore SIGPIPE, so that write()s to pipes
         * with no reader do not result in us being killed,
         * and write() returns EPIPE instead.
         */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}

/* Function that each child calls ( to be “replaced” with the executables */
void child_exec(char *exec)
{
        char *args[] = { exec, NULL, NULL, NULL };
        char *env[] = { NULL };
        printf("Inside child with PID=%d.\n", getpid());
        printf("Replacing with executable \"%s\"\n", exec);
        execve(exec, args, env);
        /* execve returns only if there is an error */
        perror("execve");
        exit(1);
}

int main(int argc, char *argv[])
{
        int nproc;
        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */
        if (argc < 2)
        {
                printf("usage: ./scheduler exec_1 exec_2 ... exec_n\n");
                exit(0);
        }
```

```c
nproc = argc - 1; /* number of proccesses */
/* Create nproc child procedures */
int i;
pid_t pids[nproc];
for(i = 0; i < nproc; i++)
{
        pids[i] = fork();
        enqueue(pids[i]);
        if(pids[i] == 0)
        {
             /* Μόλις ένα παιδί δημιουργηθεί, σταματά */
                raise(SIGSTOP);
                child_exec(argv[i+1]);
                exit(0);
        }
}

/* Wait for all children to raise SIGSTOP before exec()ing. */
wait_for_ready_children(nproc);
printf("Children processes created.\n");
print_queue();
current = dequeue();

if(current == 0)
{
        printf("Tried to dequeue from empty queue. Exiting...\n");
        exit(1);
}
/* Install SIGALRM and SIGCHLD handlers. */
install_signal_handlers();

kill(current, SIGCONT);
alarm(SCHED_TQ_SEC);

if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
}
/* loop forever  until we exit from inside a signal handler. */
while (pause())
        ;
```

```
        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

Παράδειγμα Εκτέλεσης:

```
oslabe15@skyros:~/erg4$ ./scheduler prog prog
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
My PID = 7429: Child PID = 7431 has been stopped by a signal, signo = 19
Children processes created.
Queue values:
7430 7431
Inside child with PID=7430.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 126
prog[7430]: This is message 0
prog[7430]: This is message 1
prog[7430]: This is message 2
prog[7430]: This is message 3
prog[7430]: This is message 4
prog[7430]: This is message 5
prog[7430]: This is message 6
prog[7430]: This is message 7
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
Inside child with PID=7431.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 42
prog[7431]: This is message 0
prog[7431]: This is message 1
prog[7431]: This is message 2
prog[7431]: This is message 3
prog[7431]: This is message 4
prog[7431]: This is message 5
prog[7431]: This is message 6
prog[7431]: This is message 7
prog[7431]: This is message 8
```

```
prog[7431]: This is message 9
prog[7431]: This is message 10
prog[7431]: This is message 11
prog[7431]: This is message 12
prog[7431]: This is message 13
prog[7431]: This is message 14
prog[7431]: This is message 15
prog[7431]: This is message 16
prog[7431]: This is message 17
prog[7431]: This is message 18
prog[7431]: This is message 19
prog[7431]: This is message 20
prog[7431]: This is message 21
Caught SIGALRM
My PID = 7429: Child PID = 7431 has been stopped by a signal, signo = 19
prog[7430]: This is message 8
prog[7430]: This is message 9
prog[7430]: This is message 10
prog[7430]: This is message 11
prog[7430]: This is message 12
prog[7430]: This is message 13
prog[7430]: This is message 14
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7431]: This is message 22
prog[7431]: This is message 23
prog[7431]: This is message 24
prog[7431]: This is message 25
prog[7431]: This is message 26
prog[7431]: This is message 27
prog[7431]: This is message 28
prog[7431]: This is message 29
prog[7431]: This is message 30
prog[7431]: This is message 31
prog[7431]: This is message 32
prog[7431]: This is message 33
prog[7431]: This is message 34
prog[7431]: This is message 35
prog[7431]: This is message 36
prog[7431]: This is message 37
prog[7431]: This is message 38
```

```
prog[7431]: This is message 39
prog[7431]: This is message 40
prog[7431]: This is message 41
prog[7431]: This is message 42
Caught SIGALRM
My PID = 7429: Child PID = 7431 has been stopped by a signal, signo = 19
prog[7430]: This is message 15
prog[7430]: This is message 16
prog[7430]: This is message 17
prog[7430]: This is message 18
prog[7430]: This is message 19
prog[7430]: This is message 20
prog[7430]: This is message 21
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7431]: This is message 43
prog[7431]: This is message 44
prog[7431]: This is message 45
prog[7431]: This is message 46
prog[7431]: This is message 47
prog[7431]: This is message 48
prog[7431]: This is message 49
prog[7431]: This is message 50
prog[7431]: This is message 51
prog[7431]: This is message 52
prog[7431]: This is message 53
prog[7431]: This is message 54
prog[7431]: This is message 55
prog[7431]: This is message 56
prog[7431]: This is message 57
prog[7431]: This is message 58
prog[7431]: This is message 59
prog[7431]: This is message 60
prog[7431]: This is message 61
prog[7431]: This is message 62
prog[7431]: This is message 63
Caught SIGALRM
My PID = 7429: Child PID = 7431 has been stopped by a signal, signo = 19
prog[7430]: This is message 22
prog[7430]: This is message 23
prog[7430]: This is message 24
```

```
prog[7430]: This is message 25
prog[7430]: This is message 26
prog[7430]: This is message 27
prog[7430]: This is message 28
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7431]: This is message 64
prog[7431]: This is message 65
prog[7431]: This is message 66
prog[7431]: This is message 67
prog[7431]: This is message 68
prog[7431]: This is message 69
prog[7431]: This is message 70
prog[7431]: This is message 71
prog[7431]: This is message 72
prog[7431]: This is message 73
prog[7431]: This is message 74
prog[7431]: This is message 75
prog[7431]: This is message 76
prog[7431]: This is message 77
prog[7431]: This is message 78
prog[7431]: This is message 79
prog[7431]: This is message 80
prog[7431]: This is message 81
prog[7431]: This is message 82
prog[7431]: This is message 83
prog[7431]: This is message 84
Caught SIGALRM
My PID = 7429: Child PID = 7431 has been stopped by a signal, signo = 19
prog[7430]: This is message 29
prog[7430]: This is message 30
prog[7430]: This is message 31
prog[7430]: This is message 32
prog[7430]: This is message 33
prog[7430]: This is message 34
prog[7430]: This is message 35
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7431]: This is message 85
prog[7431]: This is message 86
prog[7431]: This is message 87
```

```
prog[7431]: This is message 88
prog[7431]: This is message 89
prog[7431]: This is message 90
prog[7431]: This is message 91
prog[7431]: This is message 92
prog[7431]: This is message 93
prog[7431]: This is message 94
prog[7431]: This is message 95
prog[7431]: This is message 96
prog[7431]: This is message 97
prog[7431]: This is message 98
prog[7431]: This is message 99
My PID = 7429: Child PID = 7431 terminated normally, exit status = 0
Child with pid 7431 died.
prog[7430]: This is message 36
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 37
prog[7430]: This is message 38
prog[7430]: This is message 39
prog[7430]: This is message 40
prog[7430]: This is message 41
prog[7430]: This is message 42
prog[7430]: This is message 43
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 44
prog[7430]: This is message 45
prog[7430]: This is message 46
prog[7430]: This is message 47
prog[7430]: This is message 48
prog[7430]: This is message 49
prog[7430]: This is message 50
prog[7430]: This is message 51
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 52
prog[7430]: This is message 53
prog[7430]: This is message 54
prog[7430]: This is message 55
prog[7430]: This is message 56
```

```
prog[7430]: This is message 57
prog[7430]: This is message 58
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 59
prog[7430]: This is message 60
prog[7430]: This is message 61
prog[7430]: This is message 62
prog[7430]: This is message 63
prog[7430]: This is message 64
prog[7430]: This is message 65
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 66
prog[7430]: This is message 67
prog[7430]: This is message 68
prog[7430]: This is message 69
prog[7430]: This is message 70
prog[7430]: This is message 71
prog[7430]: This is message 72
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 73
prog[7430]: This is message 74
prog[7430]: This is message 75
prog[7430]: This is message 76
prog[7430]: This is message 77
prog[7430]: This is message 78
prog[7430]: This is message 79
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 80
prog[7430]: This is message 81
prog[7430]: This is message 82
prog[7430]: This is message 83
prog[7430]: This is message 84
prog[7430]: This is message 85
prog[7430]: This is message 86
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 87
```

```
prog[7430]: This is message 88
prog[7430]: This is message 89
prog[7430]: This is message 90
prog[7430]: This is message 91
prog[7430]: This is message 92
prog[7430]: This is message 93
Caught SIGALRM
My PID = 7429: Child PID = 7430 has been stopped by a signal, signo = 19
prog[7430]: This is message 94
prog[7430]: This is message 95
prog[7430]: This is message 96
prog[7430]: This is message 97
prog[7430]: This is message 98
prog[7430]: This is message 99
My PID = 7429: Child PID = 7430 terminated normally, exit status = 0
Child with pid 7430 died.
All done. Exiting...
```

Απαντήσεις στις ερωτήσεις:

1. Τι συμβαίνει αν το σήμα SIGALRM έρθει ενώ εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD ή το αντίστροφο; Πώς αντιμετωπίζει ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα ανάλογα ενδεχόμενα και πώς η δική σας υλοποίηση; Υπόδειξη: μελετήστε τη συνάρτηση install_signal_handlers() που δίνεται.

Όπως φαίνεται και από την install_signal_handlers(), στην δική μας υλοποίηση τα σήματα είναι με mask, οπότε αν έρθει κάποιο άλλο σήμα κατά την εκτέλεση ενός handler, αυτό απλά περιμένει, χωρίς να επηρεάζει την εκτέλεση του προγράμματος. Ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα ωστόσο χειρίζεται τα σήματα με κάποια προτεραιότητα.

2. Κάθε φορά που ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD, σε ποια διεργασία-παιδί περιμένετε να αναφέρεται αυτό; Τι συμβαίνει αν λόγω εξωτερικού παράγοντα (π.χ. αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία-παιδί;

Αναμένουμε ότι θα αναφέρεται στο παιδί που εκτελείται αυτή τη στιγμή, το οποίο είτε πέθανε (έκανε exit) ή του στείλαμε SIGSTOP. Αν όμως έρθει SIGKILL “απ’ έξω” και το παιδί πεθάνει, τότε όταν ο χρονοδρομολογητής θα χειριστεί το SIGCHLD του

στέλνοντας SIGCONT στην επόμενη διεργασία, χωρίς όμως να σταματήσει την τρέχουσα, με αποτέλεσμα να τρέχουν και οι δύο διεργασίες μαζί. Στην περίπτωση που το σήμα στέλνεται στην τρέχουσα, τότε απλά εκείνη θα πεθάνει και θα σταλεί SIGCONT στην επόμενη.

3. Γιατί χρειάζεται ο χειρισμός δύο σημάτων για την υλοποίηση του χρονοδρομολογητή; θα μπορούσε ο χρονοδρομολογητής να χρησιμοποιεί μόνο το σήμα SIGALRM για να σταματά την τρέχουσα διεργασία και να ξεκινά την επόμενη; Τι ανεπιθύμητη συμπεριφορά θα μπορούσε να εμφανίζει μια τέτοια υλοποίηση; Υπόδειξη: Η παραλαβή του σήματος SIGCHLD εγγυάται ότι η τρέχουσα διεργασία έλαβε το σήμα SIGSTOP και έχει σταματήσει.

Μια τέτοια υλοποίηση θα είχε δύο σοβαρά προβλήματα: Πρώτον, δεν θα μπορούσε να χειριστεί την περίπτωση που κάποιο παιδί κάνει exit μόνο του, και δεύτερον, θα είχε προβλήματα χρονισμού γιατί θεωρεί δεδομένου ότι όταν στέλνει SIGSTOP το παιδί σταματάει αμέσως, πράγμα που δεν είναι αλήθεια. Το παιδί για διάφορους λόγους μπορεί να καθυστερήσει λίγο να λάβει το σήμα και να σταματήσει όντως. Για να είμαστε λοιπόν σίγουροι ότι το παιδί έχει όντως σταματήσει, στέλνουμε το SIGSTOP και περιμένουμε να μας έρθει SIGCHLD που να σημαίνει ότι όντως το παιδί το έλαβε και σταμάτησε, και χειριζόμαστε ύστερα αυτό.

Άσκηση 1.2

Source Code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
```

```c
#define SCHED_TQ_SEC 5                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */

/*Global Variable*/
pid_t current, shell_pid, killed_pid;
int ids, current_id, new_node = 0;
char *current_name;
/*
 *  Queue Initialization and Functions
 */

// η ουρά αποτελείται από node του παρακάτω τύπου
typedef struct node_t
{
        int id;
        pid_t pid;
        char *name;
        struct node_t *next;
} node;

/* έχουμε θέσει ένα δείκτη που δείχνει στην αρχή της ουράς και έναν που δείχνει στο τέλος */
node *front = NULL;
node *rear = NULL;

// συνάρτηση που επιστρέφει 1 αν η ουρά είναι κενή
int isEmpty(void)
{
        if(front == NULL) return 1;
        else return 0;
}

/* συνάρτηση που προσθέτει ένα node στο τέλος της ουράς και δέχεται ως παραμέτρους τα στοιχεία του node */
void enqueue(int num, pid_t x, char *myname)
{
        node *temp = (node *)malloc(sizeof(node));
        temp->id = num;
        temp->pid = x;
        temp->name = myname;
        temp->next = NULL;
```

```
        if(front == NULL && rear == NULL)
        {
                front = rear = temp;
                return;
        }
        rear->next = temp;
        rear = temp;
}

/* συνάρτηση που διαγράφει τον πρώτο κόμβο από την αρχή της ουράς και επιστρέφει το pid του κόμβου
που διαγράφηκε */
pid_t dequeue(void)
{
        pid_t ret = 0;
        node *temp = front;
        if(front == NULL)
        {
                printf("Queue is empty!\n");
                return 0;   // Error value (Program is oriented to be used for
                            // PIDs, so this is appropriate here...)
        }
        if(front == rear)
        {
                ret = temp->pid;
                front = rear = NULL;
        }
        else
        {
                front = front->next;
                ret = temp->pid;
        }
        current_id = temp->id;
        current_name = temp->name;
        free(temp);
        return ret;
}

/* συνάρτηση που τυπώνει το περιεχόμενο της ουράς ***
πρώτα τυπώνεται η τρέχουσα διεργασία που έχει ήδη αφαιρεθεί από την ουρά και ύστερα οι υπολοιπες */
void print_queue(void)
{
```

```
        node *temp;
        printf("Queue values:\n");
        printf("ID:%d, PID:%d, NAME:%s  <-Current\n", current_id, current,
current_name);
        for(temp = front; temp != NULL; temp = temp->next)
        {
                printf("ID:%d, PID:%d, NAME:%s\n", temp->id, temp->pid, temp->name);
        }
}

/* Function that each child calls */
void child_exec(char *exec)
{
        char *args[] = { exec, NULL, NULL, NULL };
        char *env[] = { NULL };
        printf("Inside child with PID=%d.\n", getpid());
        printf("Replacing with executable \"%s\"\n", exec);
        execve(exec, args, env);
        /* execve returns only if there is an error */
        perror("execve");
        exit(1);
}


/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        print_queue();
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
        node *temp, *delete;
        temp = front;
        if (front == NULL)
        {
```

```
                printf("There is nothing to kill\n");
                return 1;
        }
        if((temp->id) == id)
        {
                /* το new_node είναι μία σημαία που αν είναι 1 σημαίνει πως έχει πεθάνει ή έχει δημιουργηθεί μία νέα διεργασία, ενώ στο killed_pid αποθηκεύουμε το pid της διεργασίας που σκοτώνουμε */
                new_node = 1;
                killed_pid = temp->pid;
                /* διαγράφουμε τον κόμβο αναζητώντας το id, και στέλνουμε SIGKILL στην αντίστοιχη διεργασία */
                front = front->next;
                if (front == NULL)
                        rear = NULL;
                kill(temp->pid, SIGKILL);
                free(temp);
                return 1;
        }
        while ((temp->next)!=NULL)
        {
                if((temp->next->id) == id)
                        break;
                temp = temp->next;
        }
        if ((temp->next)!=NULL)
        {
                new_node = 1;
                killed_pid = temp->next->pid;
                delete= temp->next;
                if ((temp->next->next) == NULL)
                        rear = temp;
                kill(temp->next->pid, SIGKILL);
                temp->next = temp->next->next;
                if ((temp->next) == NULL)
                        rear = temp;
                free(delete);
        }
        return 1;
}
```

```c
/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        pid_t p;
        new_node = 1;  /* δημιουργούμε νέα διεργασία άρα θέτουμε τη σημαία new_node=1 */
        p = fork();
        enqueue(ids, p, executable);
        ids++;
        if(p == 0)
        {
                raise(SIGSTOP);
                child_exec(executable);
                exit(0);
        }
}

/* Process requests by the shell.  */
static int
process_request(struct request_struct *rq)
{
        switch (rq->request_no) {
                case REQ_PRINT_TASKS:
                        sched_print_tasks();
                        return 0;

                case REQ_KILL_TASK:
                        return sched_kill_task_by_id(rq->task_arg);

                case REQ_EXEC_TASK:
                        sched_create_task(rq->exec_task_arg);
                        return 0;

                default:
                        return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
```

```c
static void
sigalrm_handler(int signum)
{
        if(signum == SIGALRM)
        {
                /* αν πάρρουμε σήμα SIGALRM στέλνουμε SIGSTOP στην τρέχουσα διεργασία */
                printf("Caught SIGALRM\n");
                kill(current, SIGSTOP);
                return;
        }
        else
        {
                printf("Unknown signal caught.Continuing...\n");
                return;
        }
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        pid_t p;
        int status;
        node *temp;
        for (;;)
        {
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);
                if (p < 0)
                {
                        perror("waitpid");
                        exit(1);
                }
                if (p == 0)
                        break;
                explain_wait_status(p, status);
                if (WIFEXITED(status) || WIFSIGNALED(status))
                {
                /* A child has died */
                        temp = front;
```

```
                        /* αν new_node=1 το παιδί σκοτώθηκε από το φλοιό και η συνάρτηση
τερματίζει εκεί */
                        if (new_node == 1)
                        {
                                new_node = 0;
                                printf("Child with pid %d died.\n", killed_pid);
                        }
                        else
                        {
                                 /* ελέγχουμε αν πέθανε ο φλοιός, αν όχι τότε το πρόγραμμα δεν
μπορεί να τερματίσει */
                                while (temp!=NULL)
                                {
                                        if(temp->id == 0)
                                                break;
                                        temp = temp->next;
                                }
                                if ((temp == NULL) && (shell_pid!=0))
                                {
                                        printf("Child with pid %d died.\n",
shell_pid);
                                        shell_pid = 0;
                                }
                                else
                                      /* αλλιώς τύπωσε κανονικά ποιο παιδί πέθανε */
                                        printf("Child with pid %d died.\n",
current);
                                if(isEmpty())
                                {
                                      /* αν η ουρά είναι άδεια το πρόγραμμα τερματίζει */
                                        printf("All done. Exiting...\n");
                                        exit(0);
                                }
                                else
                                {
                                      /* αφαίρεσε τον επόμενο κόμβο και στείλε σε αυτόν
SIGCONT */
                                        current = dequeue();
                                        if(current == 0)
                                        {
                                                printf("Tried to dequeue from empty
```

```
queue.Exiting...");
                                                exit(0);
                                        }
                                        kill (current, SIGCONT);
                                }
                        }
                }
                if (WIFSTOPPED(status))
                {
                 /* A child has stopped due to SIGSTOP/SIGTSTP, etc */
                  /* αν new_node=1 δημιουργήθηκε παιδί από το φλοιό οπότε η συνάρτηση τερματίζει
*/
                        if (new_node == 1)
                                new_node = 0;
                        else
                        {
                            /* αλλιώς πρόσθεσε την τρέχουσα διεργασία στο τέλος της ουράς,
αφαίρεσε την επόμενη, στείλε SIGCONT και ξεκίνα τον timer */
                                enqueue(current_id, current, current_name);
                                current = dequeue();
                                kill(current, SIGCONT);
                                alarm(SCHED_TQ_SEC);
                        }
                }
        }
}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void
signals_disable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}
```

```c
/* Enable delivery of SIGALRM and SIGCHLD.  */
static void
signals_enable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}


/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void
install_signal_handlers(void)
{
        sigset_t sigset;
        struct sigaction sa;

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

        sa.sa_handler = sigalrm_handler;
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
```

```
                exit(1);
        }

        /*
        * Ignore SIGPIPE, so that write()s to pipes
        * with no reader do not result in us being killed,
        * and write() returns EPIPE instead.
        */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}

static void
do_shell(char *executable, int wfd, int rfd)
{
        char arg1[10], arg2[10];
        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        sprintf(arg1, "%05d", wfd);
        sprintf(arg2, "%05d", rfd);
        newargv[1] = arg1;
        newargv[2] = arg2;

        raise(SIGSTOP);
        execve(executable, newargv, newenviron);

        /* execve() only returns on error */
        perror("scheduler: child: execve");
        exit(1);
}

/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static void
```

```
sched_create_shell(char *executable, int *request_fd, int *return_fd)
{
        pid_t p;
        int pfds_rq[2], pfds_ret[2];

        if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
                perror("pipe");
                exit(1);
        }

        p = fork();
        shell_pid = p;
        if (p < 0) {
                perror("scheduler: fork");
                exit(1);
        }

        if (p == 0) {
                /* Child */
                close(pfds_rq[0]);
                close(pfds_ret[1]);
                do_shell(executable, pfds_rq[1], pfds_ret[0]);

        }
        /* Parent */
       // πρόσθεσε το φλοιό στην ουρά
        enqueue(ids,p,"shell");
        ids++;
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];
}

static void
shell_request_loop(int request_fd, int return_fd)
{
        int ret;
        struct request_struct rq;

        /*
```

```
         * Keep receiving requests from the shell.
         */
        for (;;) {
                if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
                        perror("scheduler: read from shell");
                        fprintf(stderr, "Scheduler: giving up on shell request
processing.\n");
                        break;
                }

                signals_disable();
                ret = process_request(&rq);
                signals_enable();

                if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
                        perror("scheduler: write to shell");
                        fprintf(stderr, "Scheduler: giving up on shell request
processing.\n");
                        break;
                }
        }
}


int main(int argc, char *argv[])
{
        int nproc;
        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;

        /* Create the shell. */
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        /* TODO: add the shell to the scheduler's tasks */

        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */
        // τα αρχεία εισόδου πρέπει να είναι περισσότερα από 1
        if (argc<2)
        {
```

```
        printf("usage: ./scheduler exec_1 exec_2 ... exec_n\n");
        exit(0);
}

nproc = argc-1; /* number of proccesses goes here */
int i;
pid_t pids[nproc];
for(i = 0; i < nproc; i++)
{
        pids[i] = fork();
        enqueue(ids, pids[i], argv[i+1]);
        ids++;
        if(pids[i] == 0)
        {
                raise(SIGSTOP);
                child_exec(argv[i+1]);
                exit(0);
        }
}

/* Wait for all children to raise SIGSTOP before exec()ing. */
wait_for_ready_children(nproc+1);
printf("Children processes created.\n");
 // αφαίρεσε τον πρώτο κόμβο από την ουρά
current = dequeue();

if(current == 0)
{
        printf("Tried to dequeue from empty queue. Exiting...\n");
        exit(1);
}

/* Install SIGALRM and SIGCHLD handlers. */
install_signal_handlers();

kill(current, SIGCONT);
alarm(SCHED_TQ_SEC);

if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
```

```
        }

        shell_request_loop(request_fd, return_fd);

        /* Now that the shell is gone, just loop forever
         * until we exit from inside a signal handler.
         */
        while (pause())
                ;

        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

Παράδειγμα Εκτέλεσης:

```
oslabe15@skyros:~/erg4$ ./scheduler-shell prog prog
My PID = 7474: Child PID = 7475 has been stopped by a signal, signo = 19
My PID = 7474: Child PID = 7476 has been stopped by a signal, signo = 19
My PID = 7474: Child PID = 7477 has been stopped by a signal, signo = 19
Children processes created.

This is the Shell. Welcome.

Shell> p
Shell: issuing request...
Shell: receiving request return value...
Queue values:
ID:0, PID:7475, NAME:shell  <-Current
ID:1, PID:7476, NAME:prog
ID:2, PID:7477, NAME:prog
Shell> e prog
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 7474: Child PID = 7478 has been stopped by a signal, signo = 19
p
Shell: issuing request...
Shell: receiving request return value...
```

```
Queue values:
ID:0, PID:7475, NAME:shell  <-Current
ID:1, PID:7476, NAME:prog
ID:2, PID:7477, NAME:prog
ID:3, PID:7478, NAME:prog
Shell> Caught SIGALRM
My PID = 7474: Child PID = 7475 has been stopped by a signal, signo = 19
Inside child with PID=7476.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 124
prog[7476]: This is message 0
prog[7476]: This is message 1
prog[7476]: This is message 2
prog[7476]: This is message 3
prog[7476]: This is message 4
prog[7476]: This is message 5
prog[7476]: This is message 6
prog[7476]: This is message 7
prog[7476]: This is message 8
prog[7476]: This is message 9
prog[7476]: This is message 10
prog[7476]: This is message 11
prog[7476]: This is message 12
prog[7476]: This is message 13
prog[7476]: This is message 14
prog[7476]: This is message 15
prog[7476]: This is message 16
prog[7476]: This is message 17
Caught SIGALRM
My PID = 7474: Child PID = 7476 has been stopped by a signal, signo = 19
Inside child with PID=7477.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 105
prog[7477]: This is message 0
prog[7477]: This is message 1
prog[7477]: This is message 2
prog[7477]: This is message 3
prog[7477]: This is message 4
prog[7477]: This is message 5
prog[7477]: This is message 6
prog[7477]: This is message 7
```

```
prog[7477]: This is message 8
prog[7477]: This is message 9
prog[7477]: This is message 10
prog[7477]: This is message 11
prog[7477]: This is message 12
prog[7477]: This is message 13
prog[7477]: This is message 14
prog[7477]: This is message 15
prog[7477]: This is message 16
prog[7477]: This is message 17
prog[7477]: This is message 18
prog[7477]: This is message 19
prog[7477]: This is message 20
prog[7477]: This is message 21
Caught SIGALRM
My PID = 7474: Child PID = 7477 has been stopped by a signal, signo = 19
Inside child with PID=7478.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 152
prog[7478]: This is message 0
prog[7478]: This is message 1
prog[7478]: This is message 2
prog[7478]: This is message 3
prog[7478]: This is message 4
prog[7478]: This is message 5
prog[7478]: This is message 6
prog[7478]: This is message 7
prog[7478]: This is message 8
prog[7478]: This is message 9
prog[7478]: This is message 10
prog[7478]: This is message 11
prog[7478]: This is message 12
prog[7478]: This is message 13
prog[7478]: This is message 14
Caught SIGALRM
My PID = 7474: Child PID = 7478 has been stopped by a signal, signo = 19
k 2
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 7474: Child PID = 7477 was terminated by a signal, signo = 9
Child with pid 7477 died.
```

```
p
Shell: issuing request...
Shell: receiving request return value...
Queue values:
ID:0, PID:7475, NAME:shell  <-Current
ID:1, PID:7476, NAME:prog
ID:3, PID:7478, NAME:prog
Shell> Caught SIGALRM
My PID = 7474: Child PID = 7475 has been stopped by a signal, signo = 19
prog[7476]: This is message 18
prog[7476]: This is message 19
prog[7476]: This is message 20
prog[7476]: This is message 21
prog[7476]: This is message 22
prog[7476]: This is message 23
prog[7476]: This is message 24
prog[7476]: This is message 25
prog[7476]: This is message 26
prog[7476]: This is message 27
prog[7476]: This is message 28
prog[7476]: This is message 29
prog[7476]: This is message 30
prog[7476]: This is message 31
prog[7476]: This is message 32
prog[7476]: This is message 33
prog[7476]: This is message 34
prog[7476]: This is message 35
Caught SIGALRM
My PID = 7474: Child PID = 7476 has been stopped by a signal, signo = 19
prog[7478]: This is message 15
prog[7478]: This is message 16
prog[7478]: This is message 17
prog[7478]: This is message 18
prog[7478]: This is message 19
prog[7478]: This is message 20
prog[7478]: This is message 21
prog[7478]: This is message 22
prog[7478]: This is message 23
prog[7478]: This is message 24
^C
```

Απαντήσεις στις ερωτήσεις:

1. Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση, ποια εμφανίζεται πάντοτε ως τρέχουσα διεργασία στη λίστα διεργασιών (εντολή 'p'); Θα μπορούσε να μη συμβαίνει αυτό; Γιατί;

Ως τρέχουσα διεργασία στη λίστα διεργασιών εμφανίζεται πάντοτε ο φλοιός. Αυτό συμβαίνει διότι η εμφάνιση της λίστας διεργασιών γίνεται με την εντολή 'p', που συμβαίνει μόνο όταν εκτελείται ο φλοιός. Αφού και ο φλοιός υφίσταται χρονοδρομολόγηση, τότε δέχεται σήμα SIGSTOP, όπως και οι υπόλοιπες διεργασίες, και άρα η λειτουργία του αναστέλλεται μέχρι να ξανάρθει η σειρά του. Ακόμα και όταν ζητήσουμε την εντολή 'p' χωρίς να είναι η σειρά του φλοιού, αυτή θα εκτελεστεί όταν έρθει η σειρά του. Επομένως, δε γίνεται να μην εμφανίζεται ο φλοιός πάντοτε ως τρέχουσα διεργασία.

2. Γιατί είναι αναγκαίο να συμπεριλάβετε κλήσεις signals_disable(), _enable() γύρω από την συνάρτηση υλοποίησης αιτήσεων του φλοιού; Υπόδειξη: Η συνάρτηση υλοποίησης αιτήσεων του φλοιού μεταβάλλει δομές όπως η ουρά εκτέλεσης των διεργασιών.

Είναι αναγκαίο να συμπεριλάβουμε κλήσεις signals_enable() και signals_disable() γύρω από τη συνάρτηση υλοποίησης αιτήσεων του φλοιού. Αυτό συμβαίνει γιατί τόσο αυτές όσο και οι συναρτήσεις χειρισμού των σημάτων μεταβάλλουν δομές όπως η ουρά εκτέλεσης των διεργασιών, πράγμα που είναι ανεπιθύμητο.

Άσκηση 1.3

Source Code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
```

```c
#include <sys/types.h>

#include "proc-common.h"
#include "request.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 5                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */

/*Global Variable*/
pid_t shell_pid, killed_pid;
int ids, new_node = 0;

/*
 *  Queue Initialization and Functions
 */

/* η ουρά που υλοποιούμε αποτελείται από κόμβους του παρακάτω τύπου */
typedef struct node_t
{
        int id;
        pid_t pid;
        char *name;
        char *priority;
        struct node_t *next;
} node;

node current;   // το struct current δείχνει την τρέχουσα διεργασία
/* έχουμε δύο ουρές, μία για τις high προτεραιότητας διεργασίες και μία για τις low */
node *front_l = NULL, *front_h = NULL;
node *rear_l = NULL, *rear_h = NULL;

// συνάρτηση που επιστρέφει 1 όταν και οι δύο ουρές είναι άδειες
int isEmpty(void)
{
        if((front_l == NULL)&&(front_h == NULL)) return 1;
        else return 0;
}

/* συνάρτηση που προσθέτει κόμβο στο τέλος της αντίστοιχης ουράς ανάλογα με την προτεραιότητα της διεργασίας και δέχεται ως ορίσματα τα στοιχεία του κόμβου που θα προσθέσουμε */
void enqueue(int num, pid_t x, char *myname, char *mypriority)
{
        node *temp = (node *)malloc(sizeof(node));
        temp->id = num;
        temp->pid = x;
        temp->name = myname;
```

```
        temp->priority = mypriority;
        temp->next = NULL;
        if(strcmp(mypriority,"low") == 0)
        {
                if(front_l == NULL && rear_l == NULL)
                {
                        front_l = rear_l = temp;
                        return;
                }
                rear_l->next = temp;
                rear_l = temp;
        }
        else
        {
                if(front_h == NULL && rear_h == NULL)
                {
                        front_h = rear_h = temp;
                        return;
                }
                rear_h->next = temp;
                rear_h = temp;
        }
}

/* συνάρτηση που δέχεται ως όρισμα έναν χαρακτήρα που δηλώνει την προτεραιότητα της διεργασίας
που θα αφαιρέσουμε, κι ύστερα αφαιρεί τον κόμβο από την αντίστοιχη ουρά επιστρέφοντας struct με τα
στοιχεία του κόμβου που αφαιρέθηκε */
node dequeue(char pr)
{
        // pid_t ret = 0;
        node ret, *temp;
        if (pr == 'l')
        {
           /* αν ως όρισμα είναι ο χαρακτήρας 'l' τότε αφαιρούμε τον πρώτο κόμβο από τη low ουρά */
                temp = front_l;
                if(front_l == NULL)
                {
                        printf("Queue is empty!\n");
                        ret.pid = 0;
                        ret.id = -1;
                        ret.name = "NONE";
                        ret.priority = "NONE";
                        return ret;   // Error value (Program is oriented to be used
                                      // for PIDs, so this is appropriate here...)
                }
                if(front_l == rear_l)
                {
                        front_l = rear_l = NULL;
```

```
                }
                else
                {
                        front_l = front_l->next;
                }
        }
        else
        {
           /* αλλιώς αν ως όρισμα είναι ο χαρακτήρας 'h' αφαιρούμε τον πρώτο κόμβο από τη high
ουρά */
                temp = front_h;
                if(front_h == NULL)
                {
                        printf("Queue is empty!\n");
                        ret.pid = 0;
                        ret.id = -1;
                        ret.name = "NONE";
                        ret.priority = "NONE";
                        return ret;  // Error value (Program is oriented to be used for
                                     // PIDs, so this is appropriate here...)
                }
                if(front_h == rear_h)
                {
                        front_h = rear_h = NULL;
                }
                else
                {
                        front_h = front_h->next;
                }
        }
        ret.pid = temp->pid;
        ret.id = temp->id;
        ret.name = temp->name;
        ret.priority = temp->priority;

        free(temp);
        return ret;
}


/* συνάρτηση που τυπώνουμε τα περιεχόμενα των ουρών *
  * πρώτα τυπώνουμε τα στοιχεία της τρέχουσας διεργασίας που ο κόμβος της έχει αφαιρεθεί από την
ουρά *
  * ύστερα τυπώνουμε πρώτα το περιεχόμενο της high ουράς και μετά της low */
void print_queue(void)
{
        node *temp;
        printf("Queue values:\n");
```

```
        printf("ID:%d, PID:%d, NAME:%s, PRIORITY:%s  <-Current\n", current.id,
current.pid, current.name, current.priority);
        for(temp = front_h; temp != NULL; temp = temp->next)
        {
                printf("ID:%d, PID:%d, NAME:%s, PRIORITY:%s\n", temp->id, temp->pid,
temp->name, temp->priority);
        }
        for(temp = front_l; temp != NULL; temp = temp->next)
        {
                printf("ID:%d, PID:%d, NAME:%s, PRIORITY:%s\n", temp->id, temp->pid,
temp->name, temp->priority);
        }
}

/* Function that each child calls */
void child_exec(char *exec)
{
        char *args[] = { exec, NULL, NULL, NULL };
        char *env[] = { NULL };
        printf("Inside child with PID=%d.\n", getpid());
        printf("Replacing with executable \"%s\"\n", exec);
        execve(exec, args, env);
        /* execve returns only if there is an error */
        perror("execve");
        exit(1);
}


/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        print_queue();
}

/* Send SIGKILL to a task determined by the value of its
  * scheduler-specific id.
  */
static int
sched_kill_task_by_id(int id)
{
        node *temp, *delete, *front;
        int i;
        front = front_l;
       // ελέγχουμε αν οι ουρές είναι άδειες
        if ((front_l == NULL)&&(front_h ==NULL))
        {
                printf("There is nothing to kill\n");
```

```
                return 1;
        }
        /* ψάχνουμε πρώτα να βρούμε το ζητούμενο κόμβο από το id στην low ουρά και ύστερα στη
high */
        for(i=0; i<2; i++)
        {
                temp = front;
                if (temp == NULL)
                        return 1;
                // ελέγχουμε αρχικά τον πρώτο κόμβο της ουράς
                if((temp->id) == id)
                {
                        /* το new_node είναι μία σημαία που αν είναι 1 σημαίνει πως έχει πεθάνει ή
έχει δημιουργηθεί μία νέα διεργασία, ενώ στο killed_pid αποθηκεύουμε το pid της διεργασίας που
σκοτώνουμε */
                        new_node = 1;
                        killed_pid = temp->pid;
                        front = front->next;
                        if (i==0)
                        {
                                front_l = front;
                                if (front == NULL)
                                        rear_l = NULL;
                        }

                        else
                        {
                                front_h = front;
                                if (front == NULL)
                                        rear_h = NULL;
                        }
                        /* μόλις βρούμε το id στέλνουμε SIGKILL στην αντίστοιχη διεργασία,
αφαιρούμε  τον κόμβο και η συνάρτηση τερματίζει*/
                        kill(temp->pid, SIGKILL);
                        free(temp);
                        return 1;
                }
                /* ελέγχουμε και τους υπόλοιπους κόμβους της ουράς*/
                while ((temp->next)!=NULL)
                {
                        if((temp->next->id) == id)
                                break;
                        temp = temp->next;
                }
                if ((temp->next)!=NULL)
                {
                        /* μόλις βρούμε το id στέλνουμε SIGKILL στην αντίστοιχη διεργασία,
αφαιρούμε  τον κόμβο και η συνάρτηση τερματίζει*/
```

```
                        new_node = 1;
                        killed_pid = temp->next->pid;
                        delete= temp->next;
                        if ((temp->next->next) == NULL)
                        {
                                if(i==0)
                                        rear_l = temp;
                                else
                                        rear_h = temp;
                        }
                        kill(temp->next->pid, SIGKILL);
                        temp->next = temp->next->next;
                        if ((temp->next) == NULL)
                        {
                                if (i==0)
                                        rear_l = temp;
                                else
                                        rear_h = temp;
                        }
                        free(delete);
                        return 1;
                }
                front = front_h;
        }
        return 1;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        pid_t p;
       /* αφού δημιουργούμε νέα διεργασία θέτουμε new_node=1 */
        new_node = 1;
        p = fork();
        // προσθέτουμε το νέο κόμβο στο τέλος της low ουράς
        enqueue(ids, p, executable, "low");
        ids++;
        if(p == 0)
        {
                raise(SIGSTOP);
                child_exec(executable);
                exit(0);
        }
}

/* συνάρτηση που μετατρέπει μία low προτεραιότητας διεργασία σε high με βάση το id *
```

```
 * μόλις εντοπίσει το id, προσθέτει στο τέλος της high ουράς ένα νέο κόμβο με τα στοιχεία της διεργασίας και διαγράφει τον παλιό από τη low ουρά *
 * ο έλεγχος γίνεται μόνο στη low ουρά */
static void
sched_high_priority(int id)
{
        node *temp, *delete;
        temp = front_l;
        // ελέγχουμε αν η ουρά είναι άδεια
        if (front_l == NULL)
        {
                printf("No low priority process\n");
                return;
        }
        // πρώτα ελέγχουμε το id του πρώτου κόμβου της low ουράς
        if((temp->id) == id)
        {
                enqueue(temp->id, temp->pid, temp->name, "high");
                front_l = front_l->next;
                if (front_l == NULL)
                        rear_l = NULL;
                free(temp);
                return;
        }
        // αν δε το βρούμε ελέγχουμε και τους υπόλοιπους κόμβους
        while ((temp->next)!=NULL)
        {
                if((temp->next->id) == id)
                        break;
                temp = temp->next;
        }
        if ((temp->next)!=NULL)
        {
                enqueue(temp->next->id, temp->next->pid, temp->next->name, "high");
                delete = temp->next;
                if ((temp->next->next) == NULL)
                        rear_l = temp;
                temp->next = temp->next->next;
                free(delete);
                if (temp->next == NULL)
                        rear_l = temp;
        }

        return;
}

/* συνάρτηση που μετατρέπει μία high προτεραιότητας διεργασία σε low με βάση το id *
  * μόλις εντοπίσει το id, προσθέτει στο τέλος της low ουράς ένα νέο κόμβο με τα στοιχεία της διεργασίας
```

```
και διαγράφει τον παλιό από τη high ουρά *
  * ο έλεγχος γίνεται μόνο στη high ουρά */
static void
sched_low_priority(int id)
{
        node *temp, *delete;
        temp = front_h;
        // ελέγχουμε αν η ουρά είναι άδεια
        if (front_h == NULL)
        {
                printf("No high priority process\n");
                return;
        }
        // ελέγχουμε τον πρώτο κόμβο της ουράς
        if((temp->id)==id)
        {
                enqueue(temp->id, temp->pid, temp->name, "low");
                front_h = front_h->next;
                if (front_h == NULL)
                        rear_h = NULL;
                free(temp);
                return;
        }
        // αν δεν βρούμε το id ελέγχουμε και την υπόλοιπη ουρά
        while ((temp->next)!=NULL)
        {
                if((temp->next->id) == id)
                        break;
                temp = temp->next;
        }
        if((temp->next)!=NULL)
        {
                enqueue(temp->next->id, temp->next->pid, temp->next->name, "low");
                delete = temp->next;
                if ((temp->next->next) == NULL)
                        rear_h = temp;
                temp->next = temp->next->next;
                free(delete);
                if (temp->next == NULL)
                        rear_h = temp;
        }
        return;
}

/* Process requests by the shell.  */
static int
process_request(struct request_struct *rq)
{
```

```
        switch (rq->request_no) {
                case REQ_PRINT_TASKS:
                        sched_print_tasks();
                        return 0;

                case REQ_KILL_TASK:
                        return sched_kill_task_by_id(rq->task_arg);

                case REQ_EXEC_TASK:
                        sched_create_task(rq->exec_task_arg);
                        return 0;

                /* προσθέτουμε της περιπτώσεις για high και low προτεραιότητα */
                case REQ_HIGH_TASK:
                        sched_high_priority(rq->task_arg);
                        return 0;

                case REQ_LOW_TASK:
                        sched_low_priority(rq->task_arg);
                        return 0;

                default:
                        return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        if(signum == SIGALRM)
        {
                // στέλνει SIGSTOP στην τρέχουσα διεργασία
                printf("Caught SIGALRM\n");
                kill(current.pid, SIGSTOP);
                return;
        }
        else
        {
                printf("Unknown signal caught.Continuing...\n");
                return;
        }
}

/*
 * SIGCHLD handler
```

```
 */
static void
sigchld_handler(int signum)
{
        pid_t p;
        int status;
        node *temp;
        for (;;)
        {
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);
                if (p < 0)
                {
                        perror("waitpid");
                        exit(1);
                }
                if (p == 0)
                        break;
                explain_wait_status(p, status);
                if (WIFEXITED(status) || WIFSIGNALED(status))
                {
                /* A child has died */
                        temp = front_l;
                       /* αν new_node=1, τότε το παιδί σκοτώθηκε από το φλοιό και άρα η
συνάρτηση τερματίζει */
                        if (new_node == 1)
                        {
                                new_node = 0;
                                printf("Child with pid %d died.\n", killed_pid);
                        }
                        else
                        {
                            /* ελέγχουμε τη low λίστα για το αν πέθανε ο φλοιός ( ο φλοιός δεν
γίνεται ποτέ high) */
                                while (temp!=NULL)
                                {
                                        if(temp->id == 0)
                                                break;
                                        temp = temp->next;
                                }

                                if ((temp == NULL) && (shell_pid != 0))
                                {
                                        printf("Child with pid %d died.\n",
shell_pid);
                                        shell_pid = 0;
                                }
                                else
                                       /* αλλιώς τυπώνουμε ποιο παιδί πέθανε */
```

```
                                        printf("Child with pid %d died.\n",
current.pid);
                                    /* αν οι ουρές είναι άδειες το πρόγραμμα τερματίζει */
                                    if(isEmpty())
                                    {
                                        printf("All done. Exiting...\n");
                                        exit(0);
                                    }
                                    else
                                    {
                                        /* διαφορετικά αφαιρούμε τον πρώτο κόμβο από τη high
ουρά ή αν είναι άδεια τον πρώτο από τη low ουρά και στέλνουμε SIGCONT στην αντίστοιχη διεργασία */
                                        if (front_h!=NULL)
                                            current = dequeue('h');
                                        else
                                            current = dequeue('l');
                                        if(current.pid == 0)
                                        {
                                            printf("Tried to dequeue from empty
queue.Exiting...");
                                            exit(0);
                                        }
                                        kill (current.pid, SIGCONT);
                                    }
                                }
                            }
                            if (WIFSTOPPED(status))
                            {
                            /* A child has stopped due to SIGSTOP/SIGTSTP, etc*/
                                /* αν new_node=1 δημιουργήθηκε νέα διεργασία από τον φλοιό και άρα η
συνάρτηση τερματίζει */
                                if (new_node == 1)
                                    new_node = 0;
                                else
                                {
                                    /* προσθέτουμε την τρέχουσα διεργασία στο τέλος της ουράς από την
οποία προήλθε */
                                    enqueue(current.id, current.pid, current.name,
current.priority);
                                    /* αφαιρούμε τον πρώτο κόμβο από τη high ουρά ή αν είναι άδεια
τον πρώτο από τη low ουρά, στέλνουμε SIGCONT στην αντίστοιχη διεργασία και ενεργοποιούμε τον timer
*/
                                    if(front_h!=NULL)
                                        current = dequeue('h');
                                    else
                                        current = dequeue('l');
                                    kill(current.pid, SIGCONT);
                                    alarm(SCHED_TQ_SEC);
```

```
                        }
                }
        }
}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void
signals_disable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}

/* Enable delivery of SIGALRM and SIGCHLD.  */
static void
signals_enable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}


/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void
install_signal_handlers(void)
{
        sigset_t sigset;
        struct sigaction sa;

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
```

```
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

        sa.sa_handler = sigalrm_handler;
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
                exit(1);
        }

        /*
         * Ignore SIGPIPE, so that write()s to pipes
         * with no reader do not result in us being killed,
         * and write() returns EPIPE instead.
         */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}

static void
do_shell(char *executable, int wfd, int rfd)
{
        char arg1[10], arg2[10];
        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        sprintf(arg1, "%05d", wfd);
        sprintf(arg2, "%05d", rfd);
        newargv[1] = arg1;
        newargv[2] = arg2;

        raise(SIGSTOP);
        execve(executable, newargv, newenviron);

        /* execve() only returns on error */
        perror("scheduler: child: execve");
        exit(1);
}

/* Create a new shell task.
 *
```

```
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static void
sched_create_shell(char *executable, int *request_fd, int *return_fd)
{
        pid_t p;
        int pfds_rq[2], pfds_ret[2];

        if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
                perror("pipe");
                exit(1);
        }

        p = fork();
        shell_pid = p;
        if (p < 0) {
                perror("scheduler: fork");
                exit(1);
        }

        if (p == 0) {
                /* Child */
                close(pfds_rq[0]);
                close(pfds_ret[1]);
                do_shell(executable, pfds_rq[1], pfds_ret[0]);

        }
        /* Parent */
       /* ο φλοιός προστίθεται στη low ουρά */
        enqueue(ids,p,"shell", "low");
        ids++;
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];
}

static void
shell_request_loop(int request_fd, int return_fd)
{
        int ret;
        struct request_struct rq;

        /*
         * Keep receiving requests from the shell.
         */
```

```
        for (;;) {
                if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
                        perror("scheduler: read from shell");
                        fprintf(stderr, "Scheduler: giving up on shell request
processing.\n");
                        break;
                }

                signals_disable();
                ret = process_request(&rq);
                signals_enable();

                if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
                        perror("scheduler: write to shell");
                        fprintf(stderr, "Scheduler: giving up on shell request
processing.\n");
                        break;
                }
        }
}


int main(int argc, char *argv[])
{
        int nproc;
        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;

        /* Create the shell. */
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        /* TODO: add the shell to the scheduler's tasks */

        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */

        // πρέπει τα αρχεία εισόδου να είναι πάνω από 1
        if (argc<2)
        {
                printf("usage: ./scheduler exec_1 exec_2 ... exec_n\n");
                exit(0);
        }

        nproc = argc-1; /* number of proccesses goes here */
        int i;
        pid_t pids[nproc];
        for(i = 0; i < nproc; i++)
```

```
        {
                pids[i] = fork();
                enqueue(ids, pids[i], argv[i+1], "low");
                ids++;
                if(pids[i] == 0)
                {
                        raise(SIGSTOP);
                        child_exec(argv[i+1]);
                        exit(0);
                }
        }

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc+1);
        printf("Children processes created.\n");
        // αφαίρεσε τον πρώτο κόμβο
        current = dequeue('l');

        if(current.pid == 0)
        {
                printf("Tried to dequeue from empty queue. Exiting...\n");
                exit(1);
        }

        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();

        kill(current.pid, SIGCONT);
        alarm(SCHED_TQ_SEC);

        if (nproc == 0) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
        }

        shell_request_loop(request_fd, return_fd);

        /* Now that the shell is gone, just loop forever
          * until we exit from inside a signal handler.
          */
        while (pause())
                ;

        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

Παράδειγμα Εκτέλεσης:

```
oslabe15@skyros:~/erg4$ ./scheduler-shell prog prog
My PID = 7576: Child PID = 7577 has been stopped by a signal, signo = 19
My PID = 7576: Child PID = 7578 has been stopped by a signal, signo = 19
My PID = 7576: Child PID = 7579 has been stopped by a signal, signo = 19
Children processes created.

This is the Shell. Welcome.

Shell> e prog
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 7576: Child PID = 7580 has been stopped by a signal, signo = 19
h 1
Shell: issuing request...
Shell: receiving request return value...
Shell> h 2
Shell: issuing request...
Shell: receiving request return value...
Shell> l 1
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Queue values:
ID:0, PID:7577, NAME:shell, PRIORITY:low  <-Current
ID:2, PID:7579, NAME:prog, PRIORITY:high
ID:3, PID:7580, NAME:prog, PRIORITY:low
ID:1, PID:7578, NAME:prog, PRIORITY:low
Shell> Caught SIGALRM
My PID = 7576: Child PID = 7577 has been stopped by a signal, signo = 19
Inside child with PID=7579.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 113
prog[7579]: This is message 0
prog[7579]: This is message 1
prog[7579]: This is message 2
prog[7579]: This is message 3
prog[7579]: This is message 4
```

```
prog[7579]: This is message 5
prog[7579]: This is message 6
prog[7579]: This is message 7
prog[7579]: This is message 8
prog[7579]: This is message 9
prog[7579]: This is message 10
prog[7579]: This is message 11
prog[7579]: This is message 12
prog[7579]: This is message 13
prog[7579]: This is message 14
prog[7579]: This is message 15
prog[7579]: This is message 16
prog[7579]: This is message 17
prog[7579]: This is message 18
prog[7579]: This is message 19
prog[7579]: This is message 20
prog[7579]: This is message 21
prog[7579]: This is message 22
prog[7579]: This is message 23
prog[7579]: This is message 24
prog[7579]: This is message 25
prog[7579]: This is message 26
prog[7579]: This is message 27
prog[7579]: This is message 28
prog[7579]: This is message 29
prog[7579]: This is message 30
prog[7579]: This is message 31
prog[7579]: This is message 32
prog[7579]: This is message 33
prog[7579]: This is message 34
prog[7579]: This is message 35
prog[7579]: This is message 36
prog[7579]: This is message 37
prog[7579]: This is message 38
prog[7579]: This is message 39
Caught SIGALRM
My PID = 7576: Child PID = 7579 has been stopped by a signal, signo = 19
prog[7579]: This is message 40
prog[7579]: This is message 41
prog[7579]: This is message 42
prog[7579]: This is message 43
```

```
prog[7579]: This is message 44
prog[7579]: This is message 45
prog[7579]: This is message 46
prog[7579]: This is message 47
prog[7579]: This is message 48
prog[7579]: This is message 49
prog[7579]: This is message 50
prog[7579]: This is message 51
prog[7579]: This is message 52
prog[7579]: This is message 53
prog[7579]: This is message 54
prog[7579]: This is message 55
prog[7579]: This is message 56
prog[7579]: This is message 57
prog[7579]: This is message 58
prog[7579]: This is message 59
prog[7579]: This is message 60
prog[7579]: This is message 61
prog[7579]: This is message 62
prog[7579]: This is message 63
prog[7579]: This is message 64
prog[7579]: This is message 65
prog[7579]: This is message 66
prog[7579]: This is message 67
prog[7579]: This is message 68
prog[7579]: This is message 69
prog[7579]: This is message 70
prog[7579]: This is message 71
prog[7579]: This is message 72
prog[7579]: This is message 73
prog[7579]: This is message 74
prog[7579]: This is message 75
prog[7579]: This is message 76
prog[7579]: This is message 77
prog[7579]: This is message 78
Caught SIGALRM
My PID = 7576: Child PID = 7579 has been stopped by a signal, signo = 19
prog[7579]: This is message 79
prog[7579]: This is message 80
prog[7579]: This is message 81
prog[7579]: This is message 82
```

```
prog[7579]: This is message 83
prog[7579]: This is message 84
prog[7579]: This is message 85
prog[7579]: This is message 86
prog[7579]: This is message 87
prog[7579]: This is message 88
prog[7579]: This is message 89
prog[7579]: This is message 90
prog[7579]: This is message 91
prog[7579]: This is message 92
prog[7579]: This is message 93
prog[7579]: This is message 94
prog[7579]: This is message 95
prog[7579]: This is message 96
prog[7579]: This is message 97
prog[7579]: This is message 98
prog[7579]: This is message 99
My PID = 7576: Child PID = 7579 terminated normally, exit status = 0
Child with pid 7579 died.
Inside child with PID=7580.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 159
prog[7580]: This is message 0
prog[7580]: This is message 1
prog[7580]: This is message 2
prog[7580]: This is message 3
prog[7580]: This is message 4
prog[7580]: This is message 5
prog[7580]: This is message 6
prog[7580]: This is message 7
prog[7580]: This is message 8
prog[7580]: This is message 9
prog[7580]: This is message 10
prog[7580]: This is message 11
prog[7580]: This is message 12
Caught SIGALRM
My PID = 7576: Child PID = 7580 has been stopped by a signal, signo = 19
Inside child with PID=7578.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 131
prog[7578]: This is message 0
```

```
prog[7578]: This is message 1
prog[7578]: This is message 2
prog[7578]: This is message 3
prog[7578]: This is message 4
prog[7578]: This is message 5
prog[7578]: This is message 6
prog[7578]: This is message 7
prog[7578]: This is message 8
prog[7578]: This is message 9
prog[7578]: This is message 10
prog[7578]: This is message 11
prog[7578]: This is message 12
prog[7578]: This is message 13
prog[7578]: This is message 14
prog[7578]: This is message 15
prog[7578]: This is message 16
prog[7578]: This is message 17
prog[7578]: This is message 18
prog[7578]: This is message 19
prog[7578]: This is message 20
prog[7578]: This is message 21
prog[7578]: This is message 22
prog[7578]: This is message 23
prog[7578]: This is message 24
prog[7578]: This is message 25
prog[7578]: This is message 26
prog[7578]: This is message 27
prog[7578]: This is message 28
prog[7578]: This is message 29
prog[7578]: This is message 30
prog[7578]: This is message 31
prog[7578]: This is message 32
prog[7578]: This is message 33
Caught SIGALRM
My PID = 7576: Child PID = 7578 has been stopped by a signal, signo = 19
k 3
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 7576: Child PID = 7580 was terminated by a signal, signo = 9
Child with pid 7580 died.
e prog
```

```
Shell: issuing request...
Shell: receiving request return value...
Shell> My PID = 7576: Child PID = 7581 has been stopped by a signal, signo = 19
h 4
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Queue values:
ID:0, PID:7577, NAME:shell, PRIORITY:low  <-Current
ID:4, PID:7581, NAME:prog, PRIORITY:high
ID:1, PID:7578, NAME:prog, PRIORITY:low
Shell> Caught SIGALRM
My PID = 7576: Child PID = 7577 has been stopped by a signal, signo = 19
Inside child with PID=7581.
Replacing with executable "prog"
prog: Starting, NMSG = 100, delay = 76
prog[7581]: This is message 0
prog[7581]: This is message 1
prog[7581]: This is message 2
prog[7581]: This is message 3
prog[7581]: This is message 4
prog[7581]: This is message 5
prog[7581]: This is message 6
prog[7581]: This is message 7
prog[7581]: This is message 8
prog[7581]: This is message 9
prog[7581]: This is message 10
prog[7581]: This is message 11
prog[7581]: This is message 12
prog[7581]: This is message 13
^C
```

Απαντήσεις στις ερωτήσεις:

1. Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.

Λιμοκτονία διεργασιών στην υλοποίηση μας παρουσιάζεται κατά την εκτέλεση του προγράμματος αν τεθεί προτεραιότητα ‘High’ στον scheduler (ή οποιοδήποτε άλλο executable που χρονοδρομολογείται υπό τον scheduler και δεν τερματίζει ποτέ από μόνο του). Αυτό θα είχε σαν συνέπεια να εκτελείται μόνο ο φλοιός (ή/και οι άλλες διεργασίες με high priority που δεν κάνουν ποτέ exit) και όσες διεργασίες έχουν low priority δεν θα εκτελεστούν ποτέ (επειδή “περιμένουν” να τελειώσουν αυτές με το high priority), οπότε λιμοκτονούν όπως λέμε.